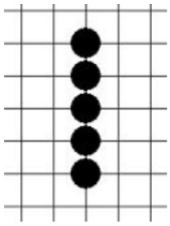
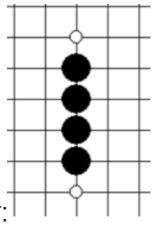
Assignment 1 report

Background

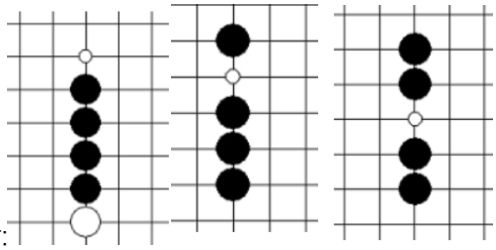
Gomoku is a classic board game where the objective is to win by placing five consecutive pieces of the same color on a 15x15 board. Before moving on to the explanation of the code detail, there are some basic patterns in Gomoku that are key concepts in the game of Gomoku.

Five: 

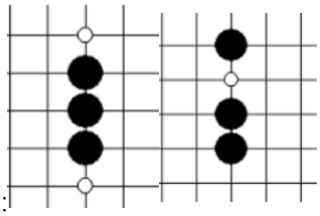
Five discs of the same color in a line, are a condition for winning this game.

Open Four: 

Four discs with two points of continuous five

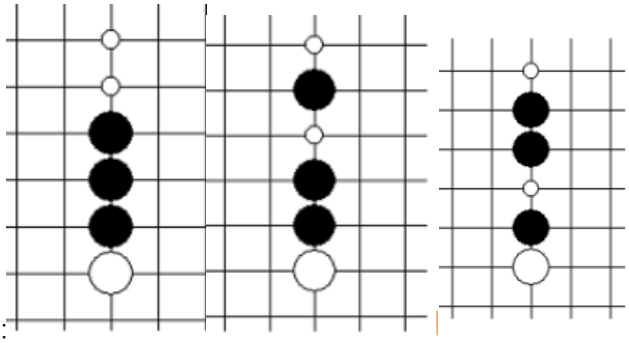
Four: 

A tetrino with one five point, i.e. only one point can form a five. Compared to the live four, the punch four is less threatening.

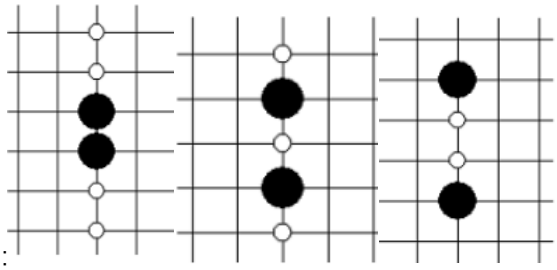
Open Three: 

Three discs that can form a live four. The live three is a very dangerous type of move,

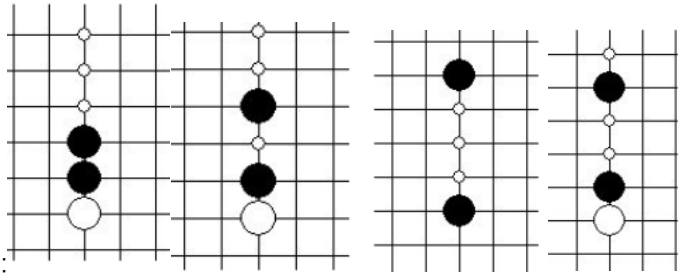
because if the opponent does not defend it, a live four can be formed on the next move.

Three: 

A trio that can only form a punch four, i.e., no matter what the next move, it can only form a punch four. Sleeping threes are relatively unthreatening because they can be defended.

Open Two: 

The live two is a two pawn shape capable of forming a live three.

Two: 

Two is a two-pawn pattern capable of forming Three.

After getting all the common chess types in Gomoku, we create a PatternType class which defines an enumeration of different pattern types that can be recognized in the game. Then we create a PATTERN_SCORES Dictionary which assigns scores to different patterns based on their value or threat level in the game. A five (five in a row) pattern has the highest score of 100,000, indicating it's a winning move. Other patterns like open_four (an open four that needs one more move to win) have a score of 10,000.

This part at the beginning of the code will be a part of the evaluation function that assesses the current state of the board and scores different patterns to determine the

best move. By scoring different patterns, the AI player can make strategic decisions based on the potential threats and opportunities present on the board.

```
class PatternType(IntEnum):
    """Pattern recognition types."""
    NONE = 0  # No pattern
    LIVE_ONE = 1  # Active one
    DEAD_ONE = 2  # Dead one
    LIVE_TWO = 3  # Active two
    DEAD_TWO = 4  # Dead two
    LIVE_THREE = 5  # Active three
    DEAD_THREE = 6  # Dead three
    LIVE_FOUR = 7  # Active four
    DEAD_FOUR = 8  # Dead four
    FIVE = 9  # Five in a row

# Modify the PATTERN_SCORES dictionary to add new patterns
PATTERN_SCORES = {
    'five': 100000,
    'open_four': 10000,
    'four': 1000,
    'open_three': 500,
    'three': 100,
    'open_two': 10,
    'two': 5,
    'double_three': 800,  # New: Double active three
    'double_four': 5000,  # New: Double active four
    'jump_four': 500,  # New: Jump four
}
```

Initialization

Below is the AdvancedTrapGomokuAI constructor initializing several attributes needed in Gomoku game.

self.max_depth = 2: Sets the maximum depth for the search algorithm to 2. This is likely used to limit the depth of the search tree to prevent the algorithm from taking too long to compute a move, which could lead to timeouts.

self.time_limit = 0.3: Sets a time limit of 0.3 seconds for the search. This is used to ensure that the AI does not spend too much time thinking about a move, which could delay the game.

self.start_time = 0: Initializes the start time to 0. This variable is likely used to track how much time has passed since the AI started thinking about its move.

self.best_move = None: Initializes the best move to None. This variable will store the best move found by the AI.

self.move_ordering_cache = {}: Initializes an empty dictionary for move ordering cache. This cache might be used to store information about the order in which moves should be considered, which can help the AI to search more efficiently.

self.player = PLAYER_A: Sets the player attribute to PLAYER_A, indicating that the AI is

representing Player A (with a value of 1).

self.opponent = PLAYER_B: Sets the opponent attribute to PLAYER_B, indicating that the AI is playing against Player B (with a value of 2).

Another function is the is_time_up which checks whether the AI has exceeded its time limit.

```
def __init__(self):
    """Initialize AI"""
    self.max_depth = 2  # Further reduce the maximum search depth to avoid
timeouts
    self.time_limit = 0.3  # Greatly reduce the time limit
    self.start_time = 0
    self.best_move = None
    self.transposition_table = {}  # Dictionary to store transposition
table
    self.move_ordering_cache = {}  # Dictionary to store move ordering
cache
    self.player = PLAYER_A  # Default to Player A (1)
    self.opponent = PLAYER_B  # Default to Player B (2)

def is_time_up(self):
    """Check if time is up"""
    current_time = time.time()
    return current_time - self.start_time > self.time_limit
```

evaluate_board(self, board)

This function evaluates the current state of the board and returns a score that indicates how favorable the board is for Player A (self.player). The higher the score, the more advantageous the board is for Player A.

It calculates the score for Player A (my_score) and Player B (opponent_score) using the _evaluate_player method and returns the final score as the difference between Player A's score and Player B's score, with Player B's score being multiplied by 1.1 to slightly favor defense.

_evaluate_player(self, board, player)

This function evaluates the score for a specified player on the current board.

It first initializes a score variable to 0 and then iterates over each cell on the board. If the cell contains the specified player's piece, it checks for patterns in all four directions. For each direction, it calls _evaluate_pattern to get the pattern score and adds it to the total score.

```
    def evaluate_board(self, board):
        """
        Evaluate the current board state
        Return a score, the higher the score, the more favorable to our
side (PLAYER_A)
        """
        # Calculate the score for our side (PLAYER_A)
        my_score = self._evaluate_player(board, self.player)

        # Calculate the score for the opponent (PLAYER_B)
        opponent_score = self._evaluate_player(board, self.opponent)

        # Return the final score: our score - opponent's score
        return my_score - opponent_score * 1.1  # Defense is slightly more
important

    def _evaluate_player(self, board, player):
        """Evaluate the score of the specified player on the current
board"""
        score = 0

        # Check each direction for patterns
        for i in range(MAX_M):
            for j in range(MAX_N):
                if board[i][j] == player:
                    # From this position, check patterns in four directions
                    for dx, dy in DIRECTIONS:
                        pattern_score = self._evaluate_pattern(board, i, j,
dx, dy, player)
                        score += pattern_score

        return score
```

_evaluate_pattern(self, board, row, col, dx, dy, player)

This function evaluates the score of a pattern starting from a specific cell and moving in a given direction.

It first constructs a pattern string by checking the board cells in the specified direction and then uses a loop to consider cells up to 4 steps away in both directions (total of 9 cells).It appends characters to the pattern string based on the cell's content: 'A' for Player A's piece, 'B' for Player B's piece, 'E' for an empty cell, 'T' for a trap, and 'O' for out of bounds. Then It calls _score_pattern with the constructed pattern string to get the score.

```
    def _evaluate_pattern(self, board, row, col, dx, dy, player):
        """
        Evaluate the pattern starting from (row,col) in the direction of
(dx,dy)
        Return the score of this pattern
        """
        # Construct the pattern string for pattern matching
        pattern = []
        for step in range(-4, 5):  # Consider 4 cells before and after,
totaling 9 cells
            r, c = row + step * dx, col + step * dy
            if 0 <= r < MAX_M and 0 <= c < MAX_N:
                if board[r][c] == player:
                    pattern.append('A')  # Our piece
                elif board[r][c] == EMPTY:
                    pattern.append('E')  # Empty space
                elif board[r][c] == TRAP:
                    pattern.append('T')  # Trap
                else:
                    pattern.append('B')  # Opponent's piece
            else:
                pattern.append('O')  # Out of bounds

        pattern_str = ''.join(pattern)
        return self._score_pattern(pattern_str, player)
```

_score_pattern(self, pattern_str, player)

This function scores a pattern string based on predefined patterns and assigns a score to a pattern string based on specific patterns that are considered advantageous or disadvantageous. Many different chess types have been listed via string, besides the trap situation is also thoroughly considered.

First, the function checks for special patterns in the given `pattern_str` that represent advantageous board configurations for the `player`. It starts by looking for a 'jump four' pattern, where a player has four pieces in a line with a gap, and if found, it returns the score for a 'jump four' from the `PATTERN_SCORES` dictionary. Then, it checks for a 'five in a row' pattern, which is a direct win condition, and returns the corresponding score. After that, the function evaluates for 'open four' and 'four in a row with one side blocked' patterns, assigning scores accordingly. Next, it looks for 'open three' and 'three in a row with one side blocked' configurations, giving them their respective scores. Following this, the function checks for 'open two' and 'two in a row with one side blocked' patterns, also assigning scores based on the `PATTERN_SCORES`. If none of these patterns are found, the function returns a score of 0, indicating a neutral or less significant board position. This systematic evaluation of patterns allows the function to assign strategic values to different move configurations, aiding the AI in making informed decisions during gameplay.

```
    def _score_pattern(self, pattern_str, player):
        # Original pattern matching remains unchanged...

        # Add a pattern for jumping four
        if 'EAEAAE' in pattern_str or 'EAAEAE' in pattern_str:
            return PATTERN_SCORES['jump_four']

        # Five in a row
        if 'AAAAA' in pattern_str:
            return PATTERN_SCORES['five']

        # Four in a row with one open end
        if 'EAAAAE' in pattern_str:
            return PATTERN_SCORES['open_four']

        # Four in a row with one side blocked
        if any(p in pattern_str for p in ['BAAAAE', 'EAAAAB', 'EAAAA',
'AAAAE','TAAAAE', 'EAAAAT']):
            return PATTERN_SCORES['four']

        # Three in a row with one open end
        if any(p in pattern_str for p in ['EAAAEE', 'EEAAAE', 'EAAEAE']):
            return PATTERN_SCORES['open_three']

        # Three in a row with one side blocked
        if any(p in pattern_str for p in ['BAAAE', 'EAAAB', 'BAEAAE',
'EAEAAB','TAEAAE', 'EAEAAT','TAAAE', 'EAAAT']):
            return PATTERN_SCORES['three']

        # Two in a row with one open end
        if any(p in pattern_str for p in ['EEAAEE', 'EAEAE']):
            return PATTERN_SCORES['open_two']

        # Two in a row with one side blocked
        if any(p in pattern_str for p in ['BAAE', 'EAAB','TAAE', 'EAAT']):
            return PATTERN_SCORES['two']

        return 0
```

get_valid_moves(self, board)

This function identifies all the valid move positions on the board where the AI can place its piece. A valid move is one that is not on a trap and not already occupied by another piece.

It iterate through the board to find all positions that are not empty (EMPTY) and add

them to occupied_positions.For each occupied position, consider the surrounding cells (up to 2 cells away in any direction). If a surrounding cell is within the boundaries and is empty, add it to the candidates set. For each candidate move, calculate its score by temporarily placing a piece there and evaluating the board state using evaluate_board. Finally, it combines the score and distance into a single score for each move, sorts the moves based on their scores and distances, prioritizing higher scores and closer distances to the center and return the top 15 moves as the most promising candidates for the AI to consider.

```
    def get_valid_moves(self, board):
        """
        Get all valid move positions (not traps, not occupied)
        """
        moves = []
        occupied_positions = set()

        # First, find all positions that already have pieces
        for i in range(MAX_M):
            for j in range(MAX_N):
                if board[i][j] in [PLAYER_A, PLAYER_B]:
                    occupied_positions.add((i, j))

        # If the board is empty, play in the center
        if not occupied_positions:
            center = MAX_M // 2
            return [(center, center)]

        # Consider empty positions around existing pieces
        candidates = set()
        for i, j in occupied_positions:
            for di in range(-2, 3):  # Consider positions within 2 cells
                for dj in range(-2, 3):
                    ni, nj = i + di, j + dj
                    if 0 <= ni < MAX_M and 0 <= nj < MAX_N and board[ni]
[nj] == EMPTY:
                        candidates.add((ni, nj))

        # Convert candidate positions to a list
        moves = list(candidates)

        # If there are no valid move positions, choose from all empty
positions
        if not moves:
            for i in range(MAX_M):
                for j in range(MAX_N):
                    if board[i][j] == EMPTY:
                        moves.append((i, j))

        # Score each possible move position for sorting
        move_scores = []
        for move in moves:
            i, j = move

            # Calculate the distance to the center
            center_dist = abs(i - MAX_M // 2) + abs(j - MAX_N // 2)

            # Evaluate the value of this position
            board[i][j] = self.player  # Try placing a piece here
            score = self.evaluate_board(board)
            board[i][j] = EMPTY     # Restore

            move_scores.append((move, score, center_dist))

        # Sort by score and center distance
        move_scores.sort(key=lambda x: (x[1], -x[2]), reverse=True)

        # Return the sorted move positions
        return [move for move, _, _ in move_scores[:15]]  # Consider only
the top 15 best positions for efficiency
```

_order_moves(self, board, moves)

This function heuristically sorts and prioritizes a list of potential moves based on various strategic criteria.

First, the function initializes a list to store potential moves along with their scores. Then, it iterates over each move, simulating the move on the board by temporarily placing the player's piece at the move's coordinates. It checks if the move would result in an immediate win for the player or block a win for the opponent, adjusting the score accordingly. Next, the function evaluates the strategic value of the move for both the player and the opponent using a quick evaluation method, combining these scores with a slight emphasis on the player's score. Additionally, a penalty is applied to moves that are farther from the center of the board. After calculating the scores for all moves, the function sorts them in descending order based on their scores. Finally, it returns the top 15 moves, providing the AI with a prioritized list of the most promising moves to consider.

```
    def _order_moves(self, board, moves):
        """Heuristically sort moves"""
        move_scores = []

        for move in moves:
            i, j = move
            score = 0

            # 1. Check if it forms a five in a row
            board[i][j] = self.player
            if self._check_win(board, i, j, self.player):
                board[i][j] = EMPTY
                return [move]  # If it can win immediately, return
immediately
            board[i][j] = EMPTY

            # 2. Check if it blocks the opponent's five in a row
            board[i][j] = self.opponent
            if self._check_win(board, i, j, self.opponent):
                score += 10000  # High score to prioritize blocking the
opponent
            board[i][j] = EMPTY

            # 3. Score based on pattern evaluation
            board[i][j] = self.player
            my_score = self._evaluate_quick(board, i, j, self.player)
            board[i][j] = self.opponent
            opp_score = self._evaluate
```

_check_win(self, board, row, col, player)

This function checks if placing a piece at the specified position (row, col) on the board would result in a winning condition for the given player, specifically a five-in-a-row.

```
def _check_win(self, board, row, col, player):
    """Check if placing a piece at (row, col) results in a five-in-a-row
win"""
    for dx, dy in DIRECTIONS:
        count = 1  # Including the current position

        # Check in one direction
        for step in range(1, 5):
            r, c = row + step * dx, col + step * dy
            if 0 <= r < MAX_M and 0 <= c < MAX_N and board[r][c] == player:
                count += 1
            else:
                break

        # Check in the opposite direction
        for step in range(1, 5):
            r, c = row - step * dx, col - step * dy
            if 0 <= r < MAX_M and 0 <= c < MAX_N and board[r][c] == player:
                count += 1
            else:
                break

        if count >= 5:
            return True

    return False
```

_evaluate_quick(self, board, row, col, player)

This function provides a quick evaluation of the strategic value of placing a piece at the specified position (row, col) on the board for the given player.

```
def _evaluate_quick(self, board, row, col, player):
    """Quickly evaluate the value of placing a piece at (row, col)"""
    score = 0
    for dx, dy in DIRECTIONS:
        pattern_score = self._evaluate_pattern(board, row, col, dx, dy,
player)
        score += pattern_score
    return score
```

negamax_alpha_beta(self, board, depth, alpha, beta, color)

This function implements the Negamax algorithm with Alpha-Beta pruning to find the best move for the AI. It is a minimax algorithm variant that considers the opponent's perspective by negating the score and acts as one of the most important algorithms in this program.

First it checks if the search has exceeded the time limit. Then generates a hash for the current board state and checks if it exists in the transposition table. If the hash exists and the depth is sufficient, return the stored score. If the search depth is 0, evaluate the board and store the result in the transposition table.

After Ggenerating valid moves for the current player, for each move, attempts to place

a piece on the board and recursively calls negamax_alpha_beta with increased depth and negated color. Updates the alpha value and checks for pruning conditions to reduce the search space. Finally store the best score and depth in the transposition table.

```
def negamax_alpha_beta(self, board, depth, alpha, beta, color):
    # Check if timeout has occurred
    if self.is_time_up():
        raise TimeoutError("Search timed out")

    # Generate a hash value for the board, used for the transposition table
    board_hash = self._hash_board(board)

    # Check the transposition table
    if board_hash in self.transposition_table and
self.transposition_table[board_hash]['depth'] >= depth:
        return self.transposition_table[board_hash]['score']

    # Termination condition: reached search depth or game over
    if depth == 0 :
        score = color * self.evaluate_board(board)
        self.transposition_table[board_hash] = {'score': score, 'depth':
depth}
        return score

    max_score = float('-inf')
    best_move = None

    # Get and sort possible move positions
    valid_moves = self.get_valid_moves(board)

    current_player = self.player if color == 1 else self.opponent

    for move in valid_moves:
        i, j = move

        # Attempt to place a piece
        board[i][j] = current_player

        # Recursive search, note that color is negated
        score = -self.negamax_alpha_beta(board, depth - 1, -beta, -alpha, -
color)

        # Restore the board
        board[i][j] = EMPTY
```

```
        # Recursive search, note that color is negated
        score = -self.negamax_alpha_beta(board, depth - 1, -beta, -alpha, -
color)

        # Restore the board
        board[i][j] = EMPTY

        # Update maximum score
        if score > max_score:
            max_score = score
            best_move = move

            # If at the top level, record the best move position
            if depth == self.max_depth:
                self.best_move = move

        # Alpha-Beta pruning
        alpha = max(alpha, max_score)
        if alpha >= beta:
            break

    # Store in the transposition table
    self.transposition_table[board_hash] = {'score': max_score, 'depth':
depth}

    return max_score
```

Open book

To let AI make a faster and more accurate decision, the opening book and the functions _is_opening and _get_opening_move provide a structured approach to handling the opening stage of the game. They offer strategic moves based on common patterns and opponent's actions. However, they have limitations in flexibility and adaptability to more complex or unique opening scenarios. Enhancing these functions to include more dynamic and adaptive strategies could further improve the AI's performance in the opening stage.

```python
def _is_opening(self, board):
    """Determine if the game is in the opening stage"""
    stone_count = 0
    for i in range(MAX_M):
        for j in range(MAX_N):
            if board[i][j] in [PLAYER_A, PLAYER_B]:
                stone_count += 1
    return stone_count <= 4  # Consider the first 4 moves as the opening stage

def _get_opening_move(self, board):
    """Retrieve a move from the opening book"""
    # Check if the board is empty
    empty = True
    for i in range(MAX_M):
        for j in range(MAX_N):
            if board[i][j] != EMPTY and board[i][j] != TRAP:
                empty = False
                break
        if not empty:
            break

    if empty:
        return OPENING_BOOK["empty"]

    # Opponent's piece is in the center
    center_x, center_y = MAX_M // 2, MAX_N // 2
    if board[center_x][center_y] == self.opponent:
        return OPENING_BOOK["center"]

    # Responses to opponent's pieces in the corners
    if board[0][0] == self.opponent:
        return OPENING_BOOK["corner_top_left"]
    if board[0][MAX_N-1] == self.opponent:
        return OPENING_BOOK["corner_top_right"]
    if board[MAX_M-1][0] == self.opponent:
        return OPENING_BOOK["corner_bottom_left"]
    if board[MAX_M-1][MAX_N-1] == self.opponent:
        return OPENING_BOOK["corner_bottom_right"]

    # Special patterns
    if board[MAX_M//2-1][MAX_N//2] == self.opponent and board[MAX_M//2+1][MAX_N//2] == self.opponent:
        return OPENING_BOOK["swap_first"]

    # Common opening lines
    if (MAX_M//2, MAX_N//2) in OPENING_BOOK["standard_line_1"]:
        if board[MAX_M//2][MAX_N//2] == self.opponent:
            return OPENING_BOOK["standard_line_1"][(MAX_M//2, MAX_N//2)]
    if (MAX_M//2, MAX_N//2) in OPENING_BOOK["standard_line_2"]:
        if board[MAX_M//2][MAX_N//2] == self.opponent:
            return OPENING_BOOK["standard_line_2"][(MAX_M//2, MAX_N//2)]

    # Respond to common shapes
    for i in range(MAX_M):
        for j in range(MAX_N):
            if board[i][j] == self.opponent and (i+1 < MAX_M and board[i+1][j+1] == self.opponent):
                return OPENING_BOOK["respond_to_diagonal"]
            if board[i][j] == self.opponent and (i+1 < MAX_M and board[i+1][j] == self.opponent):
                return OPENING_BOOK["respond_to_straight"]

    # Other opening patterns...
    return None
```

```python
OPENING_BOOK = {
    # Empty board, first move
    "empty": (7, 7),  # Center

    # Opponent is in the center, our second move
    "center": (6, 6),  # Top-left corner

    # Responses to opponent in the four corners
    "corner_top_left": (8, 8),     # Opponent at top-left (0,0), we play closer to the center
    "corner_top_right": (8, 6),    # Opponent at top-right (0,14), we play closer to the center
    "corner_bottom_left": (6, 8),  # Opponent at bottom-left (14,0), we play closer to the center
    "corner_bottom_right": (6, 6), # Opponent at bottom-right (14,14), we play closer to the center

    # Special patterns
    "swap_first": (8, 8),  # Swap first strategy, opponent around the center, we choose a symmetric position
    "block_three": None,   # Block opponent from forming a live three, needs dynamic judgment

    # Common opening lines
    "standard_line_1": {  # Standard opening line 1
        (7, 7): (6, 8),   # Opponent center, we play bottom-left
        (6, 8): (8, 6),   # Opponent bottom-left, we play top-right
        (8, 6): (8, 8)    # Opponent top-right, we play bottom-right
    },

    "standard_line_2": {  # Standard opening line 2
        (7, 7): (8, 8),   # Opponent center, we play bottom-right
        (8, 8): (6, 6),   # Opponent bottom-right, we play top-left
        (6, 6): (8, 6)    # Opponent top-left, we play top-right
    },

    # Respond to common shapes
    "respond_to_diagonal": (6, 8), # Respond to opponent's two pieces on a diagonal
    "respond_to_straight": (7, 9)  # Respond to opponent's two pieces on a straight line
}
```

find_best_move(self, board)

This function finds the best move for the AI by using a combination of quick checks, iterative deepening, and heuristic evaluations.

It first checks if there's an immediate winning move or a defensive move that needs to be made (_find_quick_win). This is a quick check to capitalize on any opportunities to win or block the opponent without delving into deeper analysis. If the game is in the opening stage (determined by _is_opening), it checks the opening book (_get_opening_move) for a predefined move. This leverages known good moves from the opening theory to start the game on a strong footing. The function then enters an iterative deepening search phase, where it gradually increases the search depth while keeping an eye on the time limit. It uses the negamax_alpha_beta function to perform a Negamax search with Alpha-Beta pruning to find the best move. This is a minimax algorithm variant that considers the opponent's perspective.

The search depth is dynamically adjusted based on the number of pieces on the board. If there are already many pieces, it reduces the search depth to avoid timeouts. Throughout the search, the function checks if the time limit is approaching. If it's about to timeout, it stops increasing the search depth to ensure it can make a move within the allotted time. If no good move is found or the search times out, the function falls back to a heuristic evaluation (_simple_evaluate) to score and choose from the valid moves. This provides a quick and simple way to select a move when time is running out.

```
def find_best_move(self, board):
    """Find the best move position"""
    # Clear cache
    self.transposition_table = {}
    self.best_move = None
    self.start_time = time.time()

    # Quick check: is there an immediate win or defensive move
    quick_win = self._find_quick_win(board)
    if quick_win:
        return quick_win

    # Check if we can use the opening book
    if self._is_opening(board):
        opening_move = self._get_opening_move(board)
        if opening_move and self._is_valid_move(board, opening_move):
            return opening_move

    # Optimization: only perform heuristic evaluation, no deep search
    # If there are already many pieces on the board, we reduce the search
depth to avoid timeouts
    piece_count = 0
    for i in range(MAX_M):
        for j in range(MAX_N):
            if board[i][j] != EMPTY:
                piece_count += 1

    # Dynamically adjust the search depth based on the number of pieces on
the board
    if piece_count > 20:
        max_depth = 2  # Reduce search depth when there are many pieces
    else:
        max_depth = min(2, self.max_depth)  # Use a lower search depth

    # Iterative deepening search with stricter time control
    try:
        for depth in range(1, max_depth + 1):
            # If more than 40% of the time has been used, do not increase
depth
            if time.time() - self.start_time > self.time_limit * 0.4:
                break
```

```
    except TimeoutError:
        # If timed out, use the previous result
        pass

    # If depth search timed out or no good move was found, use quick
heuristic method
    if self.best_move is None or not self._is_valid_move(board,
self.best_move):
        valid_moves = self.get_valid_moves(board)
        # Use a simple evaluation function
        best_score = float('-inf')
        for move in valid_moves[:min(10, len(valid_moves))]:  # Consider
only the top 10 possible moves
            if self._is_valid_move(board, move):
                x, y = move
                board[x][y] = self.player
                score = self._simple_evaluate(board, x, y)
                board[x][y] = EMPTY

                if score > best_score:
                    best_score = score
                    self.best_move = move

                # Check if about to timeout
                if time.time() - self.start_time > self.time_limit * 0.9:
                    break

    # Final safety check
    if self.best_move and not self._is_valid_move(board, self.best_move):
        # If still no valid move found, look for any empty position
        self.best_move = self._find_any_empty_position(board)

    return self.best_move
```

_simple_evaluate(self, board, x, y)

This function provides a simple evaluation of a move based on the immediate board configuration.

First, check if placing a piece at the given position results in a win. Then count the number of consecutive pieces in four main directions. Finally, assign scores based on the number of consecutive pieces and open ends.

```
def _simple_evaluate(self, board, x, y):
    """Simple board evaluation function, used to quickly assess the quality
of a move"""
    score = 0
    # Check if we can win
    if self._check_win(board, x, y, self.player):
        return 10000

    # Check eight directions
    directions = [(0, 1), (1, 0), (1, 1), (1, -1)]

    for dx, dy in directions:
        # Count the number of pieces in both directions
        count_own = 1  # Including the current position
        count_empty = 0

        # Positive direction
        nx, ny = x + dx, y + dy
        while 0 <= nx < MAX_M and 0 <= ny < MAX_N:
            if board[nx][ny] == self.player:
                count_own += 1
            elif board[nx][ny] == EMPTY:
                count_empty += 1
                break
            else:
                break
            nx, ny = nx + dx, ny + dy

        # Negative direction
        nx, ny = x - dx, y - dy
        while 0 <= nx < MAX_M and 0 <= ny < MAX_N:
            if board[nx][ny] == self.player:
                count_own += 1
            elif board[nx][ny] == EMPTY:
                count_empty += 1
                break
            else:
                break
            nx, ny = nx - dx, ny - dy
```

```
        # Scoring rules
        if count_own >= 5:
            score += 10000  # Five in a row
        elif count_own == 4 and count_empty >= 1:
            score += 1000   # Four in a row with one open end
        elif count_own == 3 and count_empty >= 2:
            score += 100    # Three in a row with two open ends
        elif count_own == 2 and count_empty >= 2:
            score += 10     # Two in a row with two open ends

    return score
```

_is_valid_move(self, board, move)

This function checks if a given move is valid by ensuring it is within the board boundaries and the position is empty.

```
def _is_valid_move(self, board, move):
    """Check if the move is valid (within bounds and position is empty)"""
    if move is None:
        return False

    i, j = move
    if 0 <= i < MAX_M and 0 <= j < MAX_N:
        return board[i][j] == EMPTY
    return False
```

play_games function

The function play_games takes three parameters: step (the current step in the game), rival_decision_x and rival_decision_y (the coordinates of the opponent's last move).It reads the current state of the board using a function read_ckbd(step-1). Then it determines which player's turn it is based on the step number, if it's not the first move and the opponent's move is within the valid range, it records the opponent's move position. Then it initializes an instance of AdvancedTrapGomokuAI, a class explained before that is designed to find the best move.

If a best move is found and the position is empty, it saves that decision.If the best move's position is not empty, it tries to find an empty position on the board and saves that decision.If no valid move is found, it defaults to placing a piece in the center of the board.

```
def play_games(step, rival_decision_x, rival_decision_y):
    # Read the current state of the board
    board = read_ckbd(step-1)

    try:
        # Determine the current player based on the step number
        current_player = PLAYER_A if step % 2 == 1 else PLAYER_B

        if step > 1 and rival_decision_x >= 0 and rival_decision_y >= 0:
            # Ensure the opponent's move is within the valid range
            if 0 <= rival_decision_x < MAX_M and 0 <= rival_decision_y <
MAX_N:
                # Record the opponent's move position, but do not modify
the board
                # because the board already contains the opponent's move
                opponent_pos = (rival_decision_x, rival_decision_y)

        ai = AdvancedTrapGomokuAI()

        if step % 2 == 1:
            ai.player = PLAYER_A
            ai.opponent = PLAYER_B
        else:
            ai.player = PLAYER_B
            ai.opponent = PLAYER_A

        best_move = ai.find_best_move(board)

        if best_move:
            x, y = best_move

            # Ensure the position is actually empty
            if board[x][y] == EMPTY:
                save_decision(x, y)
                return
            else:
                for i in range(MAX_M):
                    for j in range(MAX_N):
                        if board[i][j] == EMPTY:
                            save_decision(i, j)
                            return

        save_decision(MAX_M//2, MAX_N//2)
```

Simulate complete process

A real process of the AI player is like this.

Assume Player A (AI) starts the game Since it's the first move, it plays in the center of the board, which is a common strategy in Gomoku to maintain symmetry and flexibility. Player B makes a move Then the AI evaluates the board after Player B's move. It uses the evaluate_board function to assess the situation and determine the best response. The AI checks for immediate wins or blocking moves using _find_quick_win. Since it's

early in the game, the AI might use the opening book again to find a strategic move that counters Player B's opening move. The game continues with both players taking turns. The AI uses its find_best_move function to determine each move, considering the current state of the board, potential wins, and strategic positioning. The AI also uses the negamax_alpha_beta function for a more in-depth analysis of the board state, employing Alpha-Beta pruning to optimize the search process.

Strengths

1.The AI effectively uses pattern recognition to evaluate board positions, which is crucial for a game like Gomoku where identifying lines of pieces is key to winning.

2.Implementing Alpha-Beta pruning in the negamax_alpha_beta function helps reduce the search space by cutting off unnecessary branches, leading to faster computation times.

3.The find_best_move function uses iterative deepening to increase the search depth within the given time constraints, which is a good approach to balance between depth of search and response time.

4.The _simple_evaluate function provides a quick heuristic evaluation of moves, which is useful for sorting potential moves and focusing on the most promising ones.

5.The use of an opening book provides the AI with a set of predefined good moves for the early stages of the game, which can help establish a strong position.

Weaknesses

1.The heuristic evaluation function might be too simple and not capture all the nuances of the game state, potentially missing some strategic opportunities.

2.The time limit for each move is fixed, which might not be optimal in all situations. Dynamic adjustment based on the complexity of the position could be more effective.

3.While the AI recognizes basic patterns, there might be room for improvement in recognizing more complex or subtle patterns that could influence the game's outcome.

Summary

Overall, the AI code provides a solid foundation for playing Gomoku, with a good balance of strategic planning and efficiency. However, there are areas for potential

improvement, particularly in adapting to different game scenarios and opponents, as well as enhancing the depth and sophistication of its strategic evaluations.

Reference

The Gomoku schematic image in the report is taken from the following website https://blog.csdn.net/marble_xu/article/details/90450436.

These sites' code ideas were referenced during the initial ideation phase

https://blog.csdn.net/marble_xu/article/details/90647361?spm=1001.2014.3001.5502

https://github.com/TommyGong08/Gobang_AI/blob/main/lab2/AI.py

https://github.com/lihongxun945/gobang

All code is implemented by myself.